

# Tight Mobile Byzantine Tolerant Atomic Storage

Silvia Bonomi\*, Antonella Del Pozzo\*, Maria Potop-Butucaru†

\*Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy  
 {bonomi, delpozzo}@dis.uniroma1.it

†Université Pierre & Marie Curie (UPMC) – Paris 6, France  
 maria.potop-butucaru@lip6.fr

**Abstract.** This paper proposes the first implementation of an atomic storage tolerant to mobile Byzantine agents. Our implementation is designed for the round-based synchronous model where the set of Byzantine nodes changes from round to round. In this model we explore the feasibility of *multi-writer multi-reader* atomic register prone to various mobile Byzantine behaviors. We prove upper and lower bounds for solving the atomic storage in all the explored models. Our results, significantly different from the static case, advocate for a deeper study of the main building blocks of distributed computing while the system is prone to mobile Byzantine failures.

**Keywords:** Atomic Storage, Byzantine mobile agents, Round-based Computation.

**This paper is eligible for the Best Student Paper Award as Antonella Del Pozzo is a full time student.**

**Type:** REGULAR PAPER

## 1 Introduction

Byzantine-tolerant storage is an active research area and this problem has been studied in various settings and models (e.g. [3, 15, 10, 11] to cite just few of them). Recently, several works investigate this problem in the case where the system starts in an arbitrary state. To cope with this situation stabilizing Byzantine tolerant algorithms have been proposed in [1, 5, 7]. In all the above mentioned works the set of Byzantine processes is assumed to be static. That is, the set of nodes exhibiting a Byzantine behavior does not change during the computation.

In the current work we investigate a different fault model where Byzantines are mobile. This model captures insiders attacks or viruses propagation. In the mobile Byzantine fault model transient state corruptions, which can be abstracted as Byzantine “agents,” can move through the network and corrupt the nodes they occupy. A node occupied by a Byzantine agent will behave arbitrarily for a transient period of time. Once the Byzantine agent leaves the node, the node eventually behaves correctly. However, the Byzantine agent may “infect” another node that behaved correctly until the

infection. This models the situation where, as soon as a faulty node is repaired, another one becomes compromised.

There are two main research directions in the mobile Byzantine area: Byzantines with constrained mobility and Byzantines with unconstrained mobility. In both models the only distributed problem studied so far is the agreement problem. Byzantines with constraint mobility were studied by Buhrman *et al.* [6]. They consider that Byzantine agents move from one node to another only when protocol messages are sent (similar to how viruses would propagate).

In the case of unconstrained mobility the motion of Byzantine agents is not tight to the message exchange. Several authors investigated the agreement problem in variants of this model: [2,4,8,12,13,14]. Reischuk [13] investigate the stability/stationarity of malicious agents for a given period of time. Ostrovsky and Yung [12] introduced the notion of mobile virus and investigate an adversary that can inject and distribute faults.

Our work follows the lines opened by Garay [8]. Garay [8] and, more recently, Banu *et al.* [2] and Sasaki *et al.* [14] or Bonnet *et al.* [4] consider, in their models, that processes execute synchronous rounds composed of three phases: *send*, *receive*, *compute*. Between two consecutive rounds, Byzantine agents can move from one host to another, hence the set of faulty processes has a bounded size although its membership can change from one round to the next.

In the current work we focus four of the above discussed models, all four consider a synchronous round-based system : Garay [8], Buhrman *et al.* [6], Sasaki *et al.* [14] and Bonnet *et al.* [4]. In the Garay's model a process has the ability to detect its own infection after the Byzantine agent left it. More precisely, during the first round following the leave of the Byzantine agent, a process enters a state, called *cured*, during which it can take preventive actions to avoid sending messages that are based on a corrupted state. Garay [8] proposes in this model an algorithm that solves Mobile Byzantine Agreement provided that  $n > 6t$  (dropped later to  $n > 4f$  in [2]).

Buhrman *et al.* [6] propose a model where the motion of Byzantine agents is tight to the message exchange. In this model they prove a tight bound for Mobile Byzantine Agreement ( $n > 3t$ , where  $t$  is the maximal number of simultaneously faulty processes) and propose a time optimal protocol that matches this bound.

Bonnet *et al.* [4] investigated the same problem in a model where processes do not have the ability to detect when Byzantine agents move. However, differently from Sasaki *et al.* [14], cured processes have *control* on the messages they send. This subtle difference on the power of Byzantine agents has an impact on the bounds for solving the agreement. If in the Sasaki's model the bound on solving agreement is  $n > 6f$  in Bonnet's model it is  $n > 5f$  and this bound is proven tight.

*Our contribution.* As far as we known, our construction is the first that builds a distributed MWMR atomic memory on top of synchronous round-based servers, which communicate by message-passing, and where some of them can exhibit a Byzantine behavior induced by a mobile malicious agent. We prove first upper bounds on the number of faulty processes for four of the mobile Byzantine models cited above: Garay [8], Buhrman *et al.* [6], Sasaki *et al.* [14] and Bonnet *et al.* [4]. Then, we propose tight implementations of a atomic register in each of these models altogether with their correctness proofs. The first study focuses the model of Garay *et al.* [8], where nodes can

detect that they were previously infected by a Byzantine agent and remain silent until their state is cleaned. In this model, we implement the atomic register provided that in each round the number of Byzantine nodes (nodes occupied by a Byzantine agent),  $f$ , is less than  $n/3$  where  $n$  is the number of correct nodes in that round. The second study concerns the models of Sasaki *et al.* [14] and Bonnet *et al.* [4], where infected nodes cannot locally detect the presence or the absence of a Byzantine agent and hence can send/compute based on a corrupted state even though the mobile agent is not anymore located at that node. In both these models we implement the atomic register provided that in each round the number of Byzantine nodes  $f$  is less than  $n/4$  where  $n$  is the number of correct nodes in the round. Note that differently from the case of the agreement problem, these models have the same power in the case of atomic memory implementation. The last studied model is Buhrman *et al.* [6] where Byzantine agents move with the messages. In this model, we provide an implementation of the atomic memory provided that  $f$  is less than  $n/2$ . Note that all the above bounds are also lower bounds for the considered models.

*Paper roadmap.* The paper is organized as follows. In Section 2 we define the model of the system and the problem of MWMR atomic memory. In Section 3 we prove upper bounds on the faulty processes necessary to implement MWMR atomic memory in the following four mobile Byzantine models: Garay [8], Buhrman *et al.* [6], Sasaki *et al.* [14] and Bonnet *et al.* [4]. In Section 4 we present a generic tight algorithm that implements MWMR atomic memory parametrized function on the considered mobile Byzantine model. The correctness of the generic algorithm is proved in Section 4.2. Finally, Section 5 concludes the paper and discuss some open research directions.

## 2 Model and Problem Definition

### 2.1 System Model

We consider a distributed system composed of an arbitrary large set of clients  $\mathcal{C}$  and a set of  $n$  servers  $\mathcal{S} = \{s_1, s_2 \dots s_n\}$ . Each process in the distributed system (i.e., both servers and clients) is identified through a unique integer identifier. Servers run a distributed protocol implementing a shared memory abstraction.

**Communication model and timing assumptions.** Processes communicate through message passing. In particular, we assume that (i) each client  $c_i \in \mathcal{C}$  can communicate with every server through a broadcast primitive, (ii) servers can communicate among them through a broadcast primitive and (iii) servers can communicate with clients through point-to-point channels. We assume that communications are authenticated (i.e., given a message  $m$ , the identity of its sender cannot be forged) and reliable (i.e. messages are not created, lost or duplicated).

The system evolves in synchronous rounds. Every round is divided in three phases: (i) *send* where processes send all the messages for the current round, (ii) *receive* where processes receive all the messages sent at the beginning of the current round and (iii) *computation* where processes process received messages and prepare those that will be sent in the next round. Processes have access to the current round number via a local

variable that we usually denote by  $r$ .

**Failure model.** We assume that an arbitrary number of clients may crash while servers are affected by *mobile Byzantine failures* (MBF) [4,8,6,14]. Informally, in the mobile Byzantine failure model, faults are represented by powerful computationally unbounded agents that move arbitrarily from a server to another. When the agent is on the server, it can corrupt its local variables, force it to send arbitrary messages (potentially different from process to process) etc... However, the agent cannot corrupt the identity of the server. We assume that, in each round, at most  $f$  servers can be affected by a mobile Byzantine failure. When an agent occupies a server  $s_i$  we will say that  $s_i$  is *faulty*. When the agent leaves  $s_i$  it is said to be *cured* until it does not restore the correct internal state. If a server is neither *faulty* nor *cured* then it is said to be *correct*. We assume similar to [4,8,14] that each server has a tamper-proof memory where it safely stores the correct algorithm code. When the agent leaves a server  $s_i$  (i.e., it becomes *cured*), it recovers the correct algorithm code from the tamper-proof memory. Concerning the assumptions on agent movements and the server awareness on its *cured* state, different models have been defined. In the paper we will consider all the variants of mobile Byzantine failures [4,8,6,14]:

- **(M1)** *Garay's model* [8]. In this model, agents can move arbitrarily from a server to another at the beginning of each round (i.e. before the send phase starts). When a server is in the *cured* state it is aware of its condition and thus can remain silent to prevent the dissemination of wrong information until its code has been completely restored and its state is corrected.
- **(M2)** *Bonnet et al.'s model* [4] and **(M3)** *Sasaki et al.'s model* [14]. As in the previous model, agents can move arbitrarily from a server to another at the beginning of each round (i.e. before the send phase starts). Differently from the Garay's model, in both models it is assumed that servers do not know if they are correct or cured when the Byzantine agent moved. The main difference between these two models is that in the [14] model a cured process still acts as a Byzantine one extra round.
- **(M4)** *Buhrman's model* [6]. Differently from the previous models, agents move together with the message (i.e., with the send or broadcast operation). However, when a server is in the *cured* state it is aware of that.

## 2.2 Atomic Registers

A register is a shared variable accessed by a set of processes, i.e. clients, through two operations, namely `read()` and `write()`. Informally, the `write()` operation updates the value stored in the shared variable while the `read()` obtains the value contained in the variable (i.e. the last written value). Every operation issued on a register is, generally, not instantaneous and it can be characterized by two events occurring at its boundary: an *invocation* event and a *reply* event. These events occur at two time instants (invocation time and reply time) according to the fictional global time.

An operation  $op$  is *complete* if both the invocation event and the reply event occur (i.e. the process executing the operation does not crash between the invocation and the reply). Contrary, an operation  $op$  is said to be *failed* if it is invoked by a process that

crashes before the reply event occurs. According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution. For ease of presentation we assume the existence of a fictional global clock and the invocation time and response time of every operation are defined with respect to this fictional clock.

Given two operations  $op$  and  $op'$ , and their invocation event and reply event times ( $t_B(op)$  and  $t_B(op')$ ) and return times ( $t_E(op)$  and  $t_E(op')$ ), we say that  $op$  *precedes*  $op'$  ( $op \prec op'$ ) iff  $t_E(op) < t_B(op')$ . If  $op$  does not precede  $op'$  and  $op'$  does not precede  $op$ , then  $op$  and  $op'$  are *concurrent* ( $op || op'$ ). Given a  $write(v)$  operation, the value  $v$  is said to be written when the operation is complete.

We assume that locally any client never performs  $read()$  and  $write()$  operation concurrently. We also assume that initially the register stores a default value  $\perp$  written by a fictional  $write(\perp)$  operation happening instantaneously at round  $r_0$ . In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of register have been defined by Lamport [9]: *safe*, *regular* and *atomic*. In this paper, we will consider a Multi-Writer/Multi-Reader (MWMR) atomic register which is specified as follows:

- Termination: If a correct client invokes an operation, it eventually returns from that operation.
- Validity: A read operation returns the last value written before its invocation, or a value written by a write operation concurrent with it.
- Ordering: There exists a total order  $S$  of  $read()$  and  $write()$  operations such (i) if  $op \prec op'$  then  $op$  appears before  $op'$  in  $S$  and (ii) any  $read()$  operation returns the value  $v$  written by the last  $write()$  preceding it in  $S$ .

### 3 Upper Bounds on the number of Faults

The next theorems provide upper bounds on the number of faulty processes for the implementation of MWMR Atomic Register in the models of mobile Byzantine faults [4,8,6,14].

**Theorem 1.** *If  $n \leq 3f$ , there exists no algorithm that implements a MWMR Atomic Register in the Garay's model [8].*

**Proof** Consider that each  $read()$  operation takes at least one round to be executed and, according to the Garay's model, at the beginning of each round servers are partitioned in three sets: (i) faulty, (ii) cured and (iii) correct. Due to the assumption that we have  $f$  faulty servers in each round, we have that, cured processes, in the worse case, are  $f$  as well (i.e., the  $f$  servers that were faulty in the previous round). Thus, considering that  $n$  is at most  $3f$ , we follows that, in the worst case, at most  $f$  processes are correct. As a consequence, considering that cured servers are silent (they do not send any message), the reader will gather at most  $2f$  values and it will be not able to distinguish those that come from correct servers from those coming from faulty one.  $\square_{Theorem 1}$

**Theorem 2.** *If  $n \leq 4f$ , there exists no algorithm that implements a MWMR Atomic Register in the Sasaki's model [14].*

**Proof** The claim simply follows by considering that each `read()` operation takes at least one round to be executed and, according to the Sasaki's model, at the beginning of each round servers are partitioned in three sets: (i) faulty, (ii) cured and (iii) correct. Due to the assumption that at most  $f$  faulty servers are in each round, it follows that, cured processes, in the worst case, are  $f$  (i.e., the  $f$  servers that was faulty in the previous round). Thus, considering that  $n$  is at most  $4f$ , we have that, in the worst case, at most  $2f$  processes are correct. As a consequence, considering that cured servers act like faulty ones as well, the reader will get back at most  $4f$  values and it will be not able to distinguish which ones come from correct servers (i.e.,  $2f$  same values  $v$ ) from those coming from faulty one (i.e.,  $2f$  same values  $v'$ ).  $\square_{Theorem\ 2}$

**Theorem 3.** *If  $n \leq 4f$ , there exists no algorithm that implements a MWMR Atomic Register in the Bonnet's model [4].*

**Proof** The claim simply follows by considering that the Bonnet's model is a particular case of Sasaki model, in which cured servers act as less powerful faulty servers, forced to send the same message to all. The same reasoning as in the proof of Theorem 2 is applied.  $\square_{Theorem\ 3}$

**Theorem 4.** *If  $n \leq 2f$  there exists no algorithm that implements a MWMR Atomic Register in the Burhman's model [6].*

**Proof** The proof is similar to the static case [3]. Let us suppose by contradiction that such algorithm exists and suppose without restraining the generality that  $n = 2f$ . Let  $v$  be the value written by the last completed `write()` operation and let us assume that no other operations are concurrent with the `read()`. In this settings, when the client gets values from servers, it will receive at most  $f$  same value  $v$  from correct servers and  $f$  same values  $v'$ , with  $v' \neq v$  from faulty servers. As a consequence, the reader has no way to distinguish between the two values and we have a contradiction.  $\square_{Theorem\ 4}$

## 4 Tight MWMR Atomic Register Implementation

In this section we present a generic algorithm  $\mathcal{A}_{Areg}$  (Fig.2-1) that implements the MWMR Atomic Register in all the above presented models. In order to abstract the knowledge a server has on its state (i.e. *cured* or *correct*), we introduce the `cured_state` oracle. When invoked via `report_cured_state()` function it returns true to *cured* servers and false to others in the Garay [8] and Buhrman *et al.* [6]. In this case the oracle is said enabled. `cured_state` oracle returns always false in Sasaki *et al.* [14] or Bonnet *et al.* [4] models. In this case the oracle is said disabled.

In the following we propose a generic MWMR atomic register algorithm that is tight for all the above models by just tuning the following three parameters:  $\alpha$ ,  $\beta$  and the `cured_state` oracle status. Let denote the number of servers with respect to *faulty* servers by  $n > \alpha f$ , where  $\alpha \in \{2, 3, 4\}$  following the mobile Byzantine model. Let  $s$

be the minimal number of required occurrences of the same value in order to chose it,  $s = n - \beta f$ . Basically  $s$  has to be greater than the number of possible wrong values that *faulty* and *cured* servers can return, which is  $\beta f$ , where  $\beta \in \{1, 2\}$  depending on the model adopted for the *cured* servers.

Table 1 summarizes the above in a synthetic way.

**Table 1.**  $\mathcal{A}_{Areg}$  parameters for the four different Mobile Byzantine Failure models.

Failure model	$Mid$	$\alpha$	$\beta$	Oracle
Garay [8]	M1	3	2	enabled
Bonnet <i>et al.</i> [4]	M2	4	2	disabled
Sasaki <i>et al.</i> [14]	M3	4	2	disabled
Burhman <i>et al.</i> [6]	M4	2	1	enabled

#### 4.1 $\mathcal{A}_{Areg}$ Algorithm description

The presented algorithm exploits the round based nature of the system model. Any `write()` operation lasts one round, during which a client sends the value and all servers deliver it in the same round. Due to the synchrony assumptions no acknowledgement messages are required and the operation can terminate. If more than one `write()` operation falls in the same round then any server receives the same set of values. The one coming from the client with the highest identifier is stored, thus any server chose the same value. The `read()` operation lasts two rounds. One round to send a read request to servers and the subsequent one to gather replies. The value which occurrence is at least the threshold  $n - \beta f$  is returned.

Along with the classical `read()` and `write(v)` operations performed by clients, for maintenance purpose in each round servers echo each other their value. Thus even though at each round at most  $f$  servers may lose the value (and no `write()` operation occurs), thanks to the echoed values at the end of each round *cured* servers are able to became *correct*, having the same *correct* servers value.

*Client local variables.* Each client  $c_i$  manages the following variables:

- $to\_send_i$ : a set in which are stored messages to be sent in the next *send* phase and emptied just after.
- $reading_i$  and  $writing_i$ : two boolean variables, only the one corresponding to the current operation is set to true.
- $op\_start_i$ : a variable in which is stored the current round when a new operation starts and set to  $\perp$  when it ends.
- $rcv_i$  is a set variable (emptied at the beginning of each round), where  $c_i$  stores messages received during the current round  $r$ .
- $replies_i$ : a set in which are stored messages delivered after a read request.

*Server local variables.* Each server  $s_j$  manages the following variables:

- $value_i$ : the maintained value.

```

At the beginning of each round  $r$ 
(01)  $echo\_vals_i \leftarrow \emptyset$ ;
(02)  $current\_writes_i \leftarrow \emptyset$ ;
(03)  $cured_i \leftarrow report\_cured\_state()$ ;



---


Send Phase of round  $r$ 
(04) if ( $\neg cured_i$ )
(05)   then broadcast ECHO( $val, i$ ); % maintenance
(06)   for each  $j \in current\_reads_i$  do
(07)     send REPLY( $value_i$ ) to  $c_j$ ; % reply to read() operations started in round  $r - 1$ 
(08)   endFor
(09) endif
(10)  $current\_reads_i \leftarrow \emptyset$ ;



---


Receive Phase of round  $r$ 
(11) for each ECHO( $v, j$ ) message in  $rcv_i$  do
(12)    $echo\_vals_i \leftarrow echo\_vals_i \cup v$ ;
(13) endFor
(14) for each WRITE( $v, j$ ) message in  $rcv_i$  do
(15)    $current\_writes_i \leftarrow current\_writes_i \cup \langle v, i \rangle$ ;
(16) endFor
(17) for each READ( $j$ ) message in  $rcv_i$  do
(18)    $current\_reads_i \leftarrow current\_reads_i \cup \{j\}$ ;



---


Computation Phase of round  $r$ 
(19) if ( $current\_writes_i \neq \emptyset$ )
(20)   then let  $v$  such that  $\exists \langle v, j \rangle \in current\_writes_i \wedge j = \max_k(\langle -, k \rangle)$ ;
(21)    $value_i \leftarrow v$ ;
(22) else if ( $(\exists v \in echo\_vals_i \mid \#occurrence(v) \geq n - \beta f)$ )
(23)   then  $value_i \leftarrow v$ ;
(24) endif
(25) endif

```

**Fig. 1.**  $\mathcal{A}_{Areg}$  implementation: code executed by any server  $s_i$ .

- $rcv_i$  is a set variable (emptied at the beginning of each round), where  $s_j$  stores messages received during the current round  $r$ .
- $echo\_vals_j$ : a set (emptied at the beginning of each round), in which are stored the echoed values by servers in each round.
- $current\_writes_j$ : a set (emptied at the beginning of each round), in which are stored values that clients want to write during the current round.
- $current\_reads_j$ : a set in which are stored the identifiers of clients whose requested for a read. It is emptied after the reply to such clients.
- $cured_j$ : boolean variable set through the  $report\_cured\_state()$  event. It is set to true by the  $cured\_state$  oracle (if enabled) when  $s_j$  is in a *cured* state. Otherwise it is always false.

*Server maintenance.* For maintenance purposes, at the beginning of each round, servers exchange their stored value  $value_j$  allowing *cured* servers to become *correct* at the end of it. Thus, during the *send* phase of each round, servers broadcast the  $ECHO(val, i)$  message (Fig.1, line 05). If not new values have been written in the current round (the condition at line 19 is not verified), during the *computation* phase (Fig.1, line 22) they chose the one with at least  $n - \beta f$  occurrences. Note that in the case in which servers are aware of being in a *cured* state (Fig.1, line 04) then they avoid to send their  $value_j$ .



```

operation read():
(01)  $to\_send_i \leftarrow to\_send_i \cup \{ READ(i) \};$ 
(02)  $reading_i \leftarrow true;$ 



---


operation write( $v$ )
(03)  $to\_send_i \leftarrow to\_send_i \cup \{ WRITE(v, i) \};$ 
(04)  $writing_i \leftarrow true;$ 



---


Send Phase of round  $r$ 
(05) for each  $M() \in to\_send_i$  do broadcast  $M();$ 
(06) if ( $op\_start_i == \perp$ )
(07)   then  $op\_start_i \leftarrow r;$ 
(08) endif
(09)  $to\_send_i \leftarrow \emptyset;$ 



---


Receive Phase of round  $r$ 
(10) for each  $REPLY(v, j)$  message in  $rcv_i$  do
(11)    $replies_i \leftarrow replies_i \cup \langle v, j \rangle;$ 
(12) endFor



---


Computation Phase of round  $r$ 
(13) if ( $writing_i \wedge op\_start_i = r$ )
(14)   then  $writing_i \leftarrow false;$ 
(15)      $op\_start_i \leftarrow \perp;$ 
(16)     return write_confirmation;
(17) endif
(18) if ( $reading_i \wedge op\_start_i = r - 1$ )
(19)   then  $reading_i \leftarrow false;$ 
(20)      $op\_start_i \leftarrow \perp;$ 
(21)     let  $v$  such that  $\exists \langle v, j \rangle \in replies_i \wedge \#occurrence(v) \geq n - \beta f;$ 
(22)      $replies_i \leftarrow \emptyset;$ 
(23)     return  $v;$ 
(24) endif

```

**Fig. 2.**  $\mathcal{A}_{Areg}$  implementation: code executed by any client  $c_i$ .

*Write operation.* When a client  $c_i$  wants to write a value  $v$ , it stores in  $to\_send_i$  a message  $WRITE(v, i)$  and sets the variable  $writing_i$  to true (Fig.2, line 03-04). At the subsequent *send* phase,  $c_i$  broadcasts  $WRITE(v, i)$  to all servers, stores the current round in  $op\_start_i$  and empties the  $to\_send_i$  set (Fig.2, line 05-09). At the server side this message will be delivered within the same round during the *receive* phase and any *correct* and *cured* server  $s_j$  stores it in  $current\_writes_j$  set (Fig.1, line 14-15). At the end of the round, during the *computation* phase, if  $current\_writes_j$  is not empty then the value associated to the highest client identifier is stored in  $value_j$  (Fig. 1, line 19-21).

Back to the client side, during its *computation* phase if  $writing_i$  is true and  $op\_start_i$  is equal to the current round  $r$ , this means that during the current round  $c_i$  performed a  $write()$  operation. Since it lasts just one round then it sets  $writing_i$  to false,  $op\_start_i$  to  $\perp$  and returns the write\_confirmation to the application layer (Fig. 2, line 13-17).

*Read operation.* When a client  $c_i$  wants to read at round  $r$  then it stores in  $to\_send_i$  a message  $READ(i)$  and sets the variable  $reading_i$  to true (Fig.2, line 01-02). At the subsequent *send* phase  $c_i$  broadcasts a  $READ(i)$  message to all servers, stores the

current round  $r$  in  $op\_start_i$  and empties the  $to\_send_i$  set (Fig.2, line 05-09). Note, the check at line 06 is necessary to avoid that  $op\_start_i$  would be updated at each round. This would not be an issue for the  $write()$  operation which lasts only one round, but in the case of  $read()$  operation it would cause the loss of information about the starting round. At server side, the  $READ(i)$  message will be delivered within the same round  $r$  and any *correct* and *cured* server  $s_j$  stores the client identifier in the  $current\_reads_j$  set (Fig. 1, line 17-18).

At the start of the next round  $r + 1$ , if server  $s_j$  is not *cured* or not aware of that then it sends the message  $REPLY(value_j)$  to all the clients in  $current\_reads_j$  set, which is emptied at the end of the *send* phase (Fig. 1, line 06-10). At client side all the  $REPLY(value_j)$  are delivered and stored in the set  $replies_i$  during the *receive* phase (Fig.2, line 10-12). Now during the *computation* phase the  $reading_i$  variable is *true* and  $op\_start_i$  is storing the previous round number. Thus  $reading_i$  is set to *false*,  $op\_start_i$  is set to  $\perp$  and the value in  $replies_i$  which occurs more than  $n - \beta f$  times is returned to the application layer and  $replies_i$  is emptied (Fig. 2, line 18-24).

## 4.2 Correctness Proofs

**Lemma 1.** *Let  $\alpha_{Mi}$  and  $\beta_{Mi}$  be the parameters for each of the 4 failure models  $Mi$  as reported in Table 1 and used by the algorithm in Fig. 1-2. Let  $n > \alpha_{Mi}f$  for each failure model  $Mi$  considered. At the end of each round, at least  $n - f$  correct servers store the same value  $v$  in their  $value_i$  local variable.*

**Proof** Each non-faulty server updates its  $value_i$  local variable at the end of each round  $r$  (i) in line 21 i.e., if there exists at least a pair in the  $current\_writes_i$  local variable, or (ii) in line 23 i.e.,  $current\_writes_i$  is empty and there exist at least  $n - \beta f$  same values in  $echo\_vals_i$ .

First we prove that one of the two cases always happens and then we prove that the number of non-faulty servers storing the same values  $v$  is  $n - f$ . The  $current\_writes_i$  local variable is initialized by any non-faulty server  $s_i$  to  $\emptyset$  at the beginning of each round  $r$  (cfr. line 02) and it is updated when a  $WRITE()$  message is received by  $s_i$ <sup>1</sup>. Thus, case (i) corresponds to a scenario where at least a  $write()$  operation is executed in round  $r$  and case (ii) corresponds to a scenario where no  $write()$  is running.

- **Case (i):  $current\_writes_i \neq \emptyset$ .** In this case the claim simply follows by considering that (i) writer clients broadcast a  $WRITE(v, j)$  message in the *send* phase of round  $r$ , (ii) clients are correct so the same set of values is delivered to all servers that will apply a deterministic function to select the value  $v$  and (iii) at most  $f$  servers are faulty and may skip the update of their  $value_i$  variable.
- **Case (ii):  $current\_writes_i = \emptyset$  and line 22 is true.** In this case, the  $value_i$  variable is updated according to the values stored in  $echo\_vals_i$ . Such variable is emptied by every non-faulty process at the beginning of each round (cfr. line 01) and is

<sup>1</sup> Recall that such  $WRITE()$  message is sent by the writer client in the *send* phase of the first round starting after the  $write()$  invocation and it is delivered by any non-faulty server in the same round.

filled in when an ECHO() message is delivered. Such message is sent at least by any server, believing it is correct, at the beginning of each round. Let  $r'$  be the round in which the last write( $v$ ) operation terminated. Note that, due to above hypothesis, a write() operation always exists as we assume a fictional write happening instantaneously at round  $r_0$ . Without loss of generality, let us consider the round  $r = r' + 1$ . Due to case (i), at the end of  $r'$ , at least  $n - f$  non-faulty servers store the same value  $v$  in their local variable  $value_i$ . Thus, at the beginning of  $r' + 1$ , at least  $n - f - x$  correct servers will send an ECHO( $v, j$ ) message, where  $x$  is the number of non-faulty processes that become faulty while passing from  $r'$  to  $r$  (i.e.  $x = f$  for all the models but Burhman's one where  $x = 0$  as faulty processes move during the send phase and not at the beginning of the round). It follows that the condition in line 22 is verified if and only if  $n - f - x \geq n - \beta f$  that is true in any model. Therefore, considering that at the end of round  $r$  non-faulty servers are exactly  $n - f$ , we have that  $n - f$  processes will execute this update. Iterating the reasoning for any  $r$  the claim follows.

□<sub>Lemma 1</sub>

**Lemma 2.** *Let us consider the algorithm in Fig. 1-2. If a correct client invokes a write() operation, it eventually returns from that operation.*

**Proof** The proof simply follows by considering that, for a write() operation invoked at some round  $r$ , the write\_confirmation is generated by the client at the end of the same round just checking the value of the variables initialized at the beginning of  $r$ .

□<sub>Lemma 2</sub>

**Lemma 3.** *Let  $\alpha_{Mi}$  and  $\beta_{Mi}$  be the parameters for each of the 4 failure models  $M_i$  as reported in Table 1 and used by the algorithm in Fig. 1-2. Let  $n > \alpha_{Mi}f$  for each failure model  $M_i$  considered. If a correct client invokes a read() operation, it eventually returns from that operation.*

**Proof** Let  $c_j$  be a client invoking a read() operation at some time  $t$ . When this happens,  $c_j$  flags that a read() operation is starting and prepares a READ() message to send at the beginning of the next send phase at round  $r$ . When  $c_j$  sends such READ() message, it updates its  $op\_start_j$  variable to  $r$  and it returns from the read() operation at round  $r + 1$  if and only if it has at least  $n - \beta f$  occurrences of the same value in the  $replies_j$  set. Such  $replies_j$  is initially empty (it has been emptied at the end of the previous read() operation) and it is filled in when  $c_j$  receives a REPLY() message (line 11) that is sent at least by non-faulty servers when they receive a READ() message.

In particular, the READ() message sent by  $c_j$  will be delivered by servers during the receiving phase of round  $r$ . When this happens, any non-faulty server will execute line 18 in Figure 1 and will store the identifier of  $c_j$  in order to send a reply at the beginning of the next round  $r + 1$ . Due to Lemma 1, at the end of round  $r$ , at least  $n - f$  non-faulty servers will store the same value  $v$ . Let us note that, during the send phase of round  $r + 1$ ,  $x$  of such servers may become faulty. Thus,  $c_j$  will find a value satisfying the condition in line 21 if and only if  $n - f - x \geq n - \beta f$ . Considering that  $x \leq f$  for

all models but Burhman's one where  $x = 0$ , we have that the condition is always true and the claim follows.

□<sub>Lemma 3</sub>

**Theorem 5 (Termination).** *If a correct client invokes an operation, it eventually returns from that operation.*

**Proof** It follows directly from Lemma 2 and Lemma 3.

□<sub>Theorem 5</sub>

**Theorem 6 (Validity).** *Let  $\alpha_{Mi}$  and  $\beta_{Mi}$  be the parameters for each of the 4 failure models  $M_i$  as reported in Table 1 and used by the algorithm in Fig. 1-2. Let  $n > \alpha_{Mi}f$  for each failure model,  $M_i$ , considered. Any  $\text{read}()$  operation returns the last value written before its invocation, or a value written by a concurrent  $\text{write}()$  operation.*

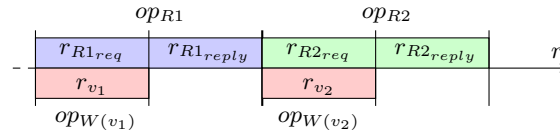
**Proof** Without loss of generality, let us consider the first  $\text{write}(v)$  operation  $op_W$  and the first  $\text{read}()$  operation  $op_R$ . Three cases may happen: (i)  $op_R \prec op_W$ , (ii)  $op_W \prec op_R$  and (iii)  $op_W \parallel op_R$ . Let us note that  $op_R$  spans over two rounds: in the first one it sends the  $\text{READ}()$  message and in the second one it collects replies.

- **Case (i):  $op_R \prec op_W$ .** This case follows directly from Lemma 1 considering that (i) at the end of the first round of  $op_R$  (i.e.,  $r_1$ ) at least  $n - f$  correct processes have the same initial value  $v = \perp$ , (ii) while moving to the second round of  $op_R$ , at most  $x$  processes can get faulty (with  $x \leq f$  for models M1-M3 and  $x = 0$  for M4), (iii)  $n - f - x \geq n - \beta_{Mi}f$  (i.e.  $\beta_{Mi}f \geq f + x$ ) for each model (i.e. there will always be enough replies from correct servers to select a value) and (iv)  $n - \beta_{Mi}f > f$  (i.e.  $(\alpha_{Mi} - \beta_{Mi})f + 1 > f$ ) for each model. It follows that faulty processes cannot force the client to select a wrong value.
- **Case (ii):  $op_W \prec op_R$ .** Let  $r$  be the round at which  $op_W$  terminates and let  $r + 1$  be the round at which  $op_R$  is invoked. Due to Lemma 1, at round  $r + 2$  there are enough occurrences (at least  $n - \beta f$ ) of the last written value  $v$ . So, applying the same reasoning of case (i) the claim follows.
- **Case (iii):  $op_W \parallel op_R$ .** Let us note that a  $\text{read}()$  operation spans two rounds, i.e., the round of the request  $r_{req}$  and the round of the reply  $r_{reply}$ . So, let us consider them separately.
  - **Case (iii-a):**  $op_W$  is concurrent with  $op_R$  during  $r_{req}$ . In that case the value  $v$  is delivered to correct server at the end of  $r_{req}$ . Due to Lemma 1, at the end of  $r_{req}$  at least  $n - f$  correct servers store the new written value  $v$ , we fall down into case (ii) and the claim follows.
  - **Case (iii-b):**  $op_W$  is concurrent with  $op_R$  during  $r_{reply}$ . Since, in every round, the send phase is executed before the receive phase, it follows that at least all the correct servers will reply with the value written before the invocation of the  $\text{write}()$  operation, we fall down into case (i) and the claim follows.

□*Theorem 6*

**Theorem 7 (Ordering).** *There exists a total order  $S$  of  $\text{read}()$  and  $\text{write}()$  operations such (i) if  $op \prec op'$  then  $op$  appears before  $op'$  in  $S$  and (ii) any  $\text{read}()$  operation returns the value  $v$  written by the last  $\text{write}()$  preceding it in  $S$ .*

**Proof** Consider two  $\text{read}()$  operations,  $op_{R1}$  and  $op_{R2}$  returning respectively  $v_1$  and  $v_2$  (with  $v_1 \neq v_2$ ) such that  $op_{R1} \prec op_{R2}$ . Note that if  $op_{R1}$  returns  $v_1$ , it follows that there exists a  $\text{write}(v_1)$  operation,  $op_{W(v_1)}$  concurrent or preceding it in  $S$ . Suppose by contradiction that  $op_{W(v_2)} \prec op_{W(v_1)}$ . Recall that each  $\text{read}()$  operation spans over two rounds and call the first  $r_{req}$  and the second  $r_{reply}$ . Since  $op_{R1}$  returns  $v_1$  this means that  $v_1$  has been stored by servers at latest during  $r_{req}$  of  $op_{R1}$ ; let us call it  $r_{R1req}$ . The same holds for  $op_{R2}$ :  $v_2$  has been written at most during  $r_{R2req}$  of  $op_{R2}$ . Since  $op_{R2}$  follows  $op_{R1}$  then  $r_{R1req} < r_{R2req}$ . However, which is a contradiction to respect the assumption of  $r_{v1} > r_{v2}$  (a general scenario is depicted in Fig.3). □*Lemma 7*



**Fig. 3.** A general scenario which show how two subsequent  $\text{read}()$  operations  $op_{R1}$  and  $op_{R2}$  can not return respectively  $v_1$  and  $v_2$  if  $v_2$  has been written before  $v_1$ .

**Theorem 8.** *Let  $\mathcal{A}_{Areg}$  be the algorithm in Fig. 1-2 and let  $n > \alpha f$ . If  $\alpha = 3$  and  $\beta = 2$  then  $\mathcal{A}_{Areg}$  implements a MWMR Atomic register in the Garay's model.*

**Proof** It follows directly from Theorem 5, 6 and 7. □*Theorem 8*

**Theorem 9.** *Let  $\mathcal{A}_{Areg}$  be the algorithm in Fig. 1-2 and let  $n > \alpha f$ . If  $\alpha = 4$  and  $\beta = 2$  then  $\mathcal{A}_{Areg}$  implements a MWMR Atomic register in the Bonnet's model.*

**Proof** It follows directly from Theorem 5, 6 and 7. □*Theorem 9*

**Theorem 10.** *Let  $\mathcal{A}_{Areg}$  be the algorithm in Fig. 1-2 and let  $n > \alpha f$ . If  $\alpha = 4$  and  $\beta = 2$  then  $\mathcal{A}_{Areg}$  implements a MWMR Atomic register in the Sasaki's model.*

**Proof** It follows directly from Theorem 5, 6 and 7. □*Theorem 10*

**Theorem 11.** *Let  $\mathcal{A}_{Areg}$  be the algorithm in Fig. 1-2 and let  $n > \alpha f$ . If  $\alpha = 2$  and  $\beta = 1$  then  $\mathcal{A}_{Areg}$  implements a MWMR Atomic register in the Burhman's model.*

**Proof** It follows directly from Theorem 5, 6 and 7. □*Theorem 11*

## 5 Conclusion

This paper addressed the first implementation of a multi-writer multi-reader atomic register tolerant to mobile Byzantine agents altogether with upper bounds on the number of faulty processes. We investigate four models of mobile Byzantines in round-based synchronous systems: the model of Garay *et al.* [8], where nodes have the capability to detect an infection and clean their state after the Byzantine agent leaves the node; the models of Sasaki *et al.* [14] and Bonnet *et al.* [4], where infected nodes may execute their code with a corrupted state even though the mobile agent is not anymore located at the node and finally, the model of Buhrman *et al.* [6] where Byzantines move are tight to messages and move during the send phase. As for the case of the agreement problem (benchmark already investigated in all these models) our study shows that the atomic registers cannot be implemented using the static bounds on the number of faulty processes. That is, we prove that in the Garay's model atomic registers can be implemented provided that in each round the number of Byzantine nodes (nodes occupied by a Byzantine agent),  $f$ , is less than  $n/3$  where  $n$  is the number of correct nodes in that round while in the Bonnet's and Sasaki's models the number of Byzantine nodes  $f$  is less than  $n/4$ . Finally, for the case of Buhrman's model we show that  $f$  should be less than  $n/2$ . Our study can be extended in several directions (here after we mention only two of them). First, an interesting issue is to investigate the storage problem in the round-free synchronous and furthermore in the asynchronous settings. We conjecture that in these models the bounds on the faulty processes are different from the round-base case. Secondly, our study advocates in favor of revisiting other building blocks of distributed computing in these settings (e.g. quorums, k-set agreement, synchronization etc). In all these cases we conjecture lower and upper bounds different from the static case.

## References

1. Alon, N., Attiya, H., Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comput. Syst. Sci.* 81(4), 692–701 (2015)
2. Banu, N., Souissi, S., Izumi, T., Wada, K.: An improved byzantine agreement algorithm for synchronous systems with mobile faults. *International Journal of Computer Applications* 43(22), 1–7 (April 2012)
3. Bazzi, R.A.: Synchronous byzantine quorum systems. *Distributed Computing* 13(1), 45–52 (Jan 2000), <http://dx.doi.org/10.1007/s004460050004>
4. Bonnet, F., Défago, X., Nguyen, T.D., Potop-Butucaru, M.: Tight bound on mobile byzantine agreement. In: *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings.* pp. 76–90 (2014)
5. Bonomi, S., Potop-Butucaru, M., Tixeuil, S.: Byzantine tolerant storage. In: *IEEE IPDPS* (2015)
6. Buhrman, H., Garay, J.A., Hoepman, J.H.: Optimal resiliency against mobile faults. In: *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS'95).* pp. 83–88 (1995)
7. Dolev, S., Dubois, S., Potop-Butucaru, M.G., Tixeuil, S.: Crash resilient and pseudo-stabilizing atomic registers. In: *OPODIS.* pp. 135–150 (2012)

8. Garay, J.A.: Reaching (and maintaining) agreement in the presence of mobile faults. In: Proceedings of the 8th International Workshop on Distributed Algorithms. vol. 857, pp. 253–264 (1994)
9. Lamport, L.: On interprocess communication. part i: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
10. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11(4), 203–213 (Oct 1998), <http://dx.doi.org/10.1007/s004460050050>
11. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: Proceedings of the 16th International Conference on Distributed Computing. pp. 311–325. DISC '02, Springer-Verlag, London, UK, UK (2002), <http://dl.acm.org/citation.cfm?id=645959.676126>
12. Ostrovsky, R., Yung, M.: How to withstand mobile virus attacks (extended abstract). In: Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91). pp. 51–59 (1991)
13. Reischuk, R.: A new solution for the byzantine generals problem. Information and Control 64(1-3), 23–42 (January-March 1985)
14. Sasaki, T., Yamauchi, Y., Kijima, S., Yamashita, M.: Mobile byzantine agreement on arbitrary network. In: Proceedings of the 17th International Conference on Principles of Distributed Systems (OPODIS'13). pp. 236–250 (December 2013)
15. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4), 299–319 (Dec 1990), <http://doi.acm.org/10.1145/98163.98167>